Lab: Half-Open Ranges with Function Templates

Learning Outcomes

- Gain experience in class templates by implementing containers that are independent from types of stored data
- Gain experience with function templates
- Gain experience in developing software that follows data abstraction and encapsulation principles
- Reading, understanding, and interpreting C++ unit tests written by others

Overview

Polymorphism is a programming technique where a single identifier, symbol, interface or construct is used in code to represent multiple different definitions. At some point the program has to determine which specific definition should be selected. We refer to polymorphism as *static*, if this decision is made at *compile time* by a compiler based on the use context. Static polymorphism has a significant cost of longer compilation times, but may contribute to zero or minimal run-time overhead. In contrast, we refer to polymorphism as *dynamic*, if the decision of which specific definition to use is made by a program at *run time*, based on execution of function calls. Dynamic polymorphism has minimal compile time cost, but may lead to slower run-time execution involving multiple memory access calls to retrieve information from a virtual table that has been generated for a type to identify functions overridden from its base type.

We have already discussed polymorphic techniques. We have covered static polymorphism disguised as function overloading, and operator overloading. Dynamic polymorphism in C++ is mostly related to inheritance and virtual functions and will be covered in future lectures this semester. Polymorphism lets us develop better abstractions: regardless of types involved, polymorphic function calls or expressions may not require syntax changes.

In this exercise we are practicing defining function and class templates, which are another manifestation of *compile time* polymorphism. Function templates and class templates represent entire families of functions or classes in a generic way – they omit specification of some internally referenced types or – as you will learn in this exercise – integral compile time numeric constants. In *generic programming*, an actual function or a class is generated from a template only when the template is used with specific parameters substituting template's type and non-type placeholders.

Task

Both C and C++ evaluate the name of a static array as a pointer to its first element. For this reason, an array name will decay to a pointer when passed to a function. Consequently the corresponding function parameter can only be defined as a pointer [to the type of array element] and will be initialized with the address of the array's first element. That function can iterate through the array elements only if the function signature is augmented with an additional parameter initialized with the array size. C++11 added the `std::array<T,N>` container to encapsulate a static array. A variable of this type has zero overhead compared to the corresponding static array because it doesn't keep any data other than the array elements. Since a variable of a user-defined type doesn't decay to a pointer when passed to a function, variables of type `std::array<T,N>` are a safer alternative to a static array.

Your task is to implement a class template that abstracts a static array - that is, a simpler clone of `std::array<T,N>`. To define a class that encapsulates a static array, you need the type of elements to be stored in the static array, and a compile time constant representing the static array's size. These parameters will not be hardcoded in your implementation. Instead, you must implement a class template, and use template type parameter `T` [for the array element type] and nontype parameter `N` [for the array dimension], respectively. The class template must have the following declaration:

```
namespace hlp2 {

template <typename T, size_t N>
class Array;
  ...
}

} // end namespace hlp2
```

You are familiar with *template type parameters* specified by keyword `typename` [from class lectures and previous week's lab]. Nontype parameters work in a similar fashion, but have a few significant differences. They represent compile time numeric constants, not types. They can only use integral types [`short`, `int`, `size_t`, `bool`, and so on], enumeration types, pointer and reference types. Floating-point types will be supported only from C++20 onwards.

Instantiations of templates with different template parameters represent different classes. Therefore, types `hlp2::Array<int, 4>` and `hlp2::Array<int, 3>` are distinct concrete classes that are instantiated from the same class template.

The lecture handout on template nontype parameters provides a good introduction to this topic. See the example code from the lecture on class lecture for a practical example of implementing a class template using both template type and nontype parameters.

> ***It is important for you to know that*** `std::array<T,N>` ***has unique semantics compared to other containers in the standard library. Unlike other containers such as*** `std::string` ***and*** `std::vector<T>`, `std::array<T,N>` ***fulfils the requirements of an aggregate. According to the C++ standard, an aggregate is an array or a class with no user-provided constructors and no private non-static data members. The standard further says that an aggregate has no base classes and no virtual functions [this is not relevant to this exercise - we'll study inheritance in a few weeks].***

Implementation details

You must implement an interface for container `hlp2::Array<T,N>` that consists of these 16 member functions: `begin()`, `begin() const`, `end()`, `end() const`, `cbegin() const`, `cend() const`, `front()`, `front() const`, `back()`, `back() const`, `operator[]()`, `operator[]() const`, `empty() const`, `size() const`, `fill()`, and `swap()`. Each of these functions should have the exact same behavior as the corresponding member function of container `std::array<T,N>`. This is a good place to study the interface provided by `std::array<T,N>` and many code examples.

While implementing the class template you should pay particular attention to the following:

1. Copy semantics, meaning how objects of such types are created or passed by copy. Ask yourself whether you need to follow the Rule of 3 here. Ask yourself whether you need to worry about memory leaks?

2. Templates' compilation. From the previous lab, you now know that definitions of function templates must be available to any translation unit that instantiates these templates. Hence, you declared and defined the function templates in a single `.hpp` file that was then included in every source file that instantiates these templates. The same is true with definitions of class templates and corresponding member functions. This week, you'll physically split the definition of class template `hlp2::Array<T,N>` and its implementation: you'll define the class template in file `array.hpp`; you'll provide the definitions of member functions in file `array.tpp`; and then you'll include `array.tpp` in file `array.hpp`:

```
#ifndef ARRAY_HPP_
#define ARRAY_HPP_

namespace hlp2 {

// definition of class template here ...

// ALL member functions must be defined in separate file array.tpp
#include "array.tpp"

} // end namespace hlp2

#endif // end ARRAY_HPP_
```

3. An important detail to remember about templates is that ***code is instantiated only for template [member] functions that are called***. That is, member functions for a class template are instantiated only if the concrete class [instantiated from the class template] calls these member functions. This, of course, saves time [during compilation] and space [for the executable generated by the linker] and allows use of class templates only partially. So one way for you to test your implementation is to define class template `hlp2::Array<T,N>` in `array.hpp` and define the complete interface in `array.tpp`. Then, write a simple driver that tests one member function at a time. Once this member function is tested, add another member function to test. This incremental approach will allow you to easily identify and fix an incorrect member function.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Header file and documentation details

You will be submitting file `array.hpp` and `array.tpp`. File `array.hpp` should have the following format:

```
//------------------------------------------------------------------------
#ifndef ARRAY_HPP
#define ARRAY _HPP
//------------------------------------------------------------------------
#include <cstdef> // for size_t
// don't include <array>!!!
// otherwise, your code won't compile.

namespace hlp2 {

```

```
// define class template Array<T,N> and document each member function

// this file will contain the definition of ALL member functions
// declared in class template Array<T,N>
#include "array.tpp"
} // end namespace hlp2

#endif
```

Compiling, executing, and testing

Download `array-driver.cpp` and output files containing the correct output for the 3 unit tests in the driver. Follow the steps from the previous lab to refactor a `makefile` that can compile, link, and test your program.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit the necessary files.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - F grade if your submission doesn't compile with the full suite of `g++` options.
    - F grade if your submission doesn't link to create an executable.
    - Your implementation's output doesn't match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). The auto grader will assign 50% of the grade based on the input and output files given to you. The remaining 50% of the grade will be awarded based on the additional tests implemented by the auto grader.
    - The auto grade will provide a proportional grade based on how many incorrect results were generated by your submission. $A+$ grade if your output matches correct output of auto grader.
    - A deduction of one letter grade for each missing documentation block in your submissions. Each source file must have **one** file-level documentation block and a function-level documentation block for each defined function. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to F.